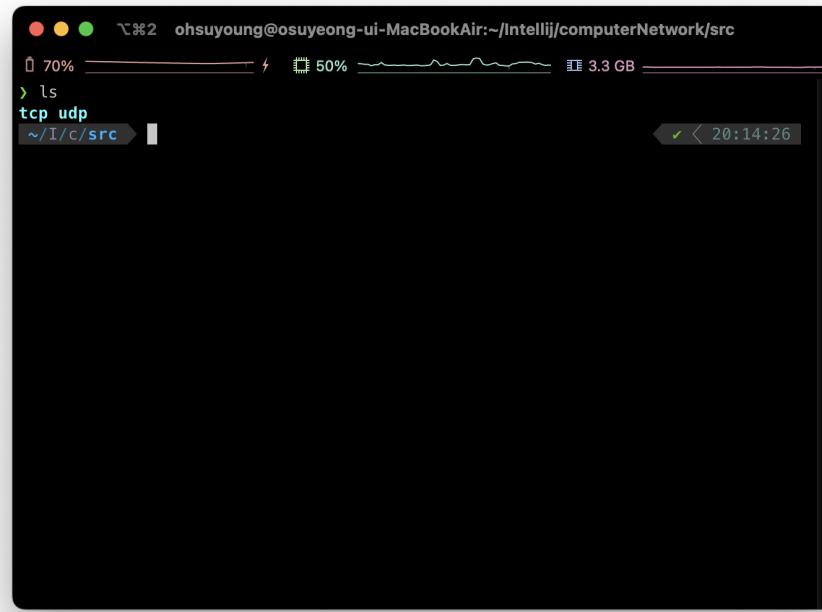


# 2020083854\_오수영\_Assignment2

컴퓨터소프트웨어학부 2020083854 오수영

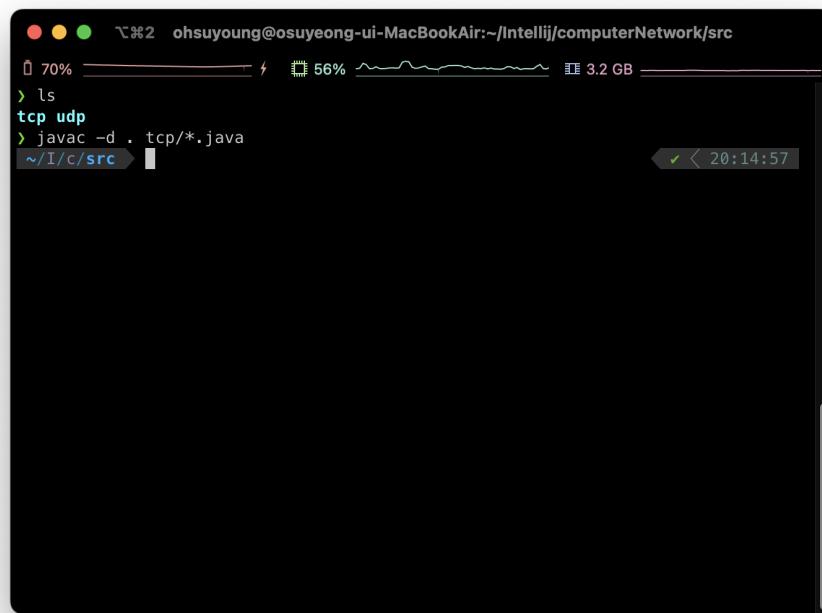
## 0. 컴파일 및 실행 방법

1. ls 명령어를 쳤을 때 tcp package를 확인할 수 있게 압축을 해제한다.



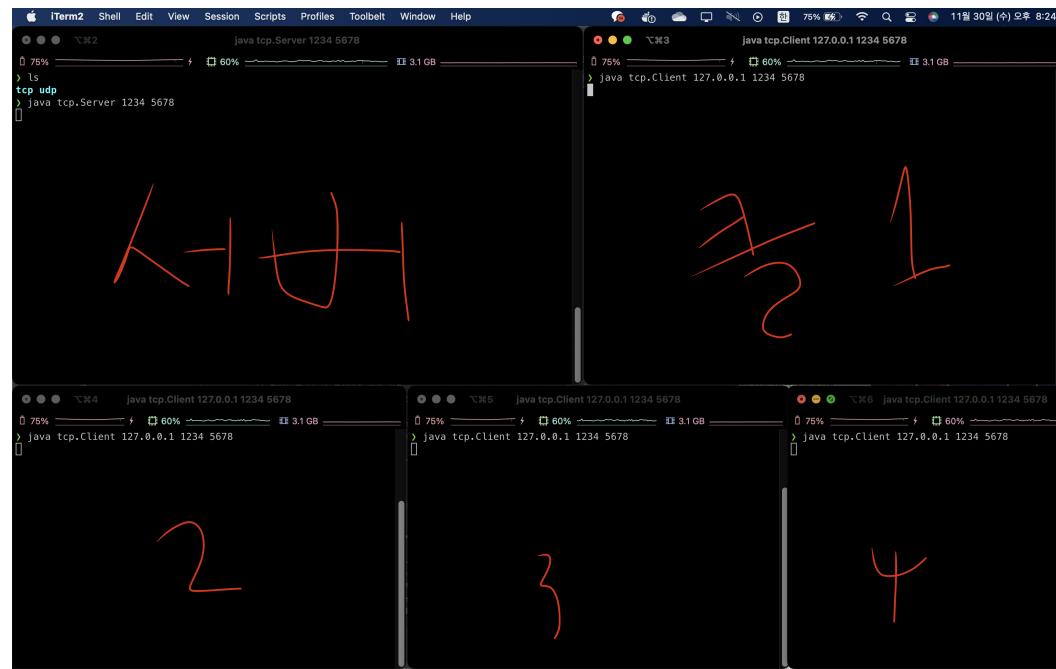
A screenshot of a macOS terminal window titled 'ohsuyeong@osuyeong-ui-MacBookAir:~/IntelliJ/computerNetwork/src'. The window shows a file tree with two main folders: 'tcp' and 'udp'. The current directory is indicated by a blue arrow pointing to 'src'. The status bar at the bottom right shows the date and time as '20:14:26'.

- javac -d . tcp/\*.java



A screenshot of a macOS terminal window titled 'ohsuyeong@osuyeong-ui-MacBookAir:~/IntelliJ/computerNetwork/src'. The user has run the command 'javac -d . tcp/\*.java'. The status bar at the bottom right shows the date and time as '20:14:57'.

2. tcp는 패키지 명이며 Server와 Client 마다 치는 명령어가 다르다



- 서버 : java tcp.Server 1234 5678 (포트 번호는 임의지정)
- 클라이언트 : java tcp.Client 127.0.0.1 1234 5678
- 위 화면이 나오면 준비가 완료된 상태이다

## 1. Server 클래스

1. main 클래스 : 매개변수로 포트번호 2개를 넣어주고 각각 소켓을 만들어준다. 생성자로 두 소켓을 넣어주고, `runServer()`에서 서버를 돌릴 예정이다.

```
public static void main(String[] args) throws IOException {
    // 서버와 채팅 메세지를 주고 받는 용도, #로 시작하는 명령어 전송
    int portNo1 = Integer.parseInt(args[0]);
    // #PUT #GET 의 동작만을 위해 사용
    int portNo2 = Integer.parseInt(args[1]);
    ServerSocket serverSocket1 = new ServerSocket(portNo1);
    ServerSocket serverSocket2 = new ServerSocket(portNo2);
    Server server = new Server(serverSocket1, serverSocket2);
    server.runServer();
}
```

2. `runServer()` : 첫번째 포트번호는 바로 accept를 해주고 두번째 포트번호는 (나중 #PUT, GET을 위해) 서버소켓만 넘겨준다. 클라이언트들은 `ClientHandler`에서 스레드로 처리할 예정이다.

- `accept()`로 클라이언트로부터의 접속을 대기한다.
- 접속에 성공하면 ClientHandler스레드를 실행시킨다.

```
public void runServer() {
    try {
        while (!serverSocket1.isClosed() && !serverSocket2.isClosed()) {
            // 접속 대기
            Socket socket = serverSocket1.accept();
            System.out.println("New client connected!");

            ClientHandler clientHandler = new ClientHandler(socket, serverSocket2);
            Thread thread = new Thread(clientHandler);
            thread.start();

        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

## 2. 클래스 `ClientHandler implements Runnable`

1. `ClientHandler`의 생성자에선 첫번째 소켓을 this로 담아주고, input과 output stream을 설정해준다. 아직 `serverSocketForFile` 사용하지 않았다.

## 2. client 가 입력한 값(msgFromClient)을 읽어들여서 명령어를 실행한다

### 2-1. msgFromClient == null 일때 소켓을 닫아준다

- split을 하여 #이후 채팅방이름이나 클라이언트이름, 파일이름들을 msgFromClients에 담아준다

```
@Override  
public void run() {  
    String msgFromClient;  
    while (!socket.isClosed()) {  
        try {  
            msgFromClient = bufferedReader.readLine();  
            if (msgFromClient == null) {  
                System.out.println("null");  
                closeEverything(socket, bufferedReader, bufferedWriter);  
                break;  
            }  
            if (msgFromClient.charAt(0) == '#') {  
                String[] msgFromClients = msgFromClient.split("\\s");
```

### 2-2. #STATUS

- Pair를 직접 구현하여 채팅방 이름과 클라이언트 이름을 한 쌍으로 묶어서 이를 key로 HashMap에 저장한다. value는 ClientHandler 객체이다.
- clientHandlers를 돌면서 this.chatRoomName이 같은 모든 클라이언트를 chatRoomInfo에 추가한 후 #STATUS를 친 클라이언트에게 개인적으로 정보를 보내준다. (sendPersonalMessage는 추후에 설명)

```
if (msgFromClients[0].equals("#STATUS")) {  
    String chatRoomInfo = "chatRoomName : "+chatRoomName +", clients [ ";  
    for (Pair<String, String> key : clientHandlers.keySet()) {  
        if (key.first.equals(this.chatRoomName))  
            chatRoomInfo = chatRoomInfo + key.second +" ";  
    }  
    chatRoomInfo += "]";  
    sendPersonalMessage(chatRoomInfo);  
}
```

### 2-3. #CREATE : 이미 만들어진 채팅방은 다시 만들 수 없음에 초점을 맞춘다.

- 해쉬맵을 돌면서 key의 first인 채팅방 이름이 존재하면 isRoomExists = true; 으로 표시해준다
- 이미 존재하면 FAIL 메세지를 클라이언트에게 보내주고
- 없는 채팅방이면 this에 지정해주고 clientHandlers에 넣어주며 SUCCESS 메세지를 전송한다.

```
else if (msgFromClients[0].equals("#CREATE")) {  
    for (Pair<String, String> key : clientHandlers.keySet()) {  
        // 해쉬맵 돌면서 채팅방이름이 같으면  
        if (key.first.equals(msgFromClients[1])) {  
            isRoomExists = true;  
            break;  
        }  
    }  
    if (isRoomExists) {  
        // 이미 존재  
        sendPersonalMessage("FAIL: chatroom [ " + msgFromClients[1] + " ] is already exists.");  
        isRoomExists = false;  
    }  
    else {  
        // 새 채팅방 생성  
        this.chatRoomName = msgFromClients[1];  
        this.clientName = msgFromClients[2];  
  
        clientHandlers.put(Pair.of(msgFromClients[1], msgFromClients[2]), this);  
  
        // 성공메세지 전송  
        sendPersonalMessage("SUCCESS: chatroom [ "+chatRoomName+" ] is successfully created.");  
    }  
}
```

### 2-4. #JOIN : 이미 존재하는 채팅방 밖에 들어갈 수 없음에 초점을 맞춘다.

- 해쉬맵을 돌면서 채팅방 이름이 같은 것이 있어야 성공이다. 그러면 this로 지정해주고 `clientHandlers`에 넣어준다. 그 후 SUCCESS 메세지를 보내준다.
- 존재하지 않는 채팅방이면 FAIL 메세지를 보내준다.

```

else if (msgFromClients[0].equals("#JOIN")) {
    for (Pair<String, String> key : clientHandlers.keySet()) {
        // 해쉬맵 돌면서 채팅방이름이 같으면
        if (key.first.equals(msgFromClients[1])) {
            isRoomExists = true;
            break;
        }
    }
    if (isRoomExists) {
        // 있는 채팅방이여야 성공
        this.chatRoomName = msgFromClients[1];
        this.clientName = msgFromClients[2];

        clientHandlers.put(Pair.of(msgFromClients[1], msgFromClients[2]), this);

        // 성공메세지 전송
        sendPersonalMessage("SUCCESS: client [ "+clientName+" ] successfully joins chat room [ "+chatRoomName+" ].");
        isRoomExists = false;
    } else {
        // 없는 채팅방에 join 하면 실패메세지 전송
        sendPersonalMessage("FAIL: chatroom [ " + msgFromClients[1] + " ] doesn't exists.");
    }
}
}

```

## 2-5. #PUT

- 이 안에서 `serverSocketForFile.accept()`를 해줘서 socket을 생성한 후
- 파일이므로 `FileOutputStream`을 이용하여 `InputStream`으로 `accept`한 소켓으로부터 `input`스트림을 가져온다.
- 64000바이트 단위로 가져오며 `fileOutputStream.write`로 써준다.
- 모두 성공하면 스트림을 닫아줌으로써 끝났음을 확실히 한 후 `broadcastMessage`로 채팅방에 있는 사람에게 누가 무슨 이름의 파일을 보냈는지 공지해준다.

```

else if (msgFromClients[0].equals("#PUT")) {
    Socket socketForFile = this.serverSocketForFile.accept();
    fileName = msgFromClients[1];

    FileOutputStream fileOutputStream = new FileOutputStream(fileName);
    InputStream fileInputStream = socketForFile.getInputStream();
    int l;
    int byteSize = 64000;
    byte[] data = new byte[byteSize];
    while ((l = fileInputStream.read(data)) != -1) {
        fileOutputStream.write(data, 0, l);
        fileOutputStream.flush();
    }

    fileInputStream.close();
    fileOutputStream.close();
    socketForFile.close();

    broadcastMessage("SERVER : [ " + clientName + " ] sends file [ " + fileName + " ]");
}

```

## 2-6. #GET

- 마찬가지로 여기서 `serverSocketForFile.accept()`로 소켓을 만들어준다.
- `fileInputStream`과 `OutputStream`을 이용하여 `get`을 요청한 클라이언트에게 전달한다.
- 마찬가지로 입출력스트림과 소켓을 닫아준다.

```

else if (msgFromClients[0].equals("#GET")) {
    Socket socketForFile = this.serverSocketForFile.accept();
    fileName = msgFromClients[1];
}

```

```

FileInputStream fileInputStream = new FileInputStream(fileName);
OutputStream fileOutputStream = socketForFile.getOutputStream();

int l;
int byteSize = 64000;
byte[] data = new byte[byteSize];
while ((l = fileInputStream.read(data)) > 0) {
    fileOutputStream.write(data, 0, l);
    fileOutputStream.flush();
}
fileInputStream.close();
fileOutputStream.close();
socketForFile.close();
}

```

## 2-7. #EXIT

- exit이면 `removeClientHandler`로 해쉬맵에서 지워줘야한다.
- #로 시작하는 것이 아니면 모두 그냥 메세지로 간주하여 `broadcastMessage(msgFromClient)` 채팅방의 모든 멤버에게 메세지를 전송한다.

```

else if (msgFromClients[0].equals("#EXIT")) {
    removeClientHandler();
}
} else {
    // 메세지 전송
    broadcastMessage(msgFromClient);
}

```

## 3. 각종 함수들

3-1. `sendPersonalMessage` : 서버에게 무엇인가를 보낸 클라이언트에게만 메세지를 전달한다

```

public void sendPersonalMessage(String message) {
    try {
        bufferedWriter.write(message);
        bufferedWriter.newLine();
        bufferedWriter.flush();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

3-2. `broadcastMessage` : 같은 채팅방의 모든 클라이언트들에게 메세지를 전달한다

```

public void broadcastMessage(String message) {
    for (Pair<String, String> key : clientHandlers.keySet()) {
        try {
            // 채팅방 이름은 같고, 클라이언트 이름은 다를 때
            if (key.first.equals(chatRoomName) && !key.second.equals(clientName)) {
                clientHandlers.get(key).bufferedWriter.write(message);
                clientHandlers.get(key).bufferedWriter.newLine();
                clientHandlers.get(key).bufferedWriter.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

3-3. `removeClientHandler()` 는 채팅방과 클라이언트이름이 존재하며 해쉬맵에서 찾을 수 있을 때만 지워주는 역할을 한다. #CREATE 보다 #JOIN이 먼저 들어왔을 때 만들지 않은 이름들을 지울 수는 없기 때문이다. `closeEverything` 은 `socket`, `bufferedReader`, `bufferedWriter` 를 close 해준다. `socket`을 먼저 close해줘야한다.

```

public void removeClientHandler() {
    if (this.chatRoomName != null && this.clientName != null && clientHandlers.containsKey(Pair.of(this.chatRoomName, this.clientName)))
    {

```

```

clientHandlers.remove(Pair.of(this.chatRoomName, this.clientName));
    broadcastMessage("SERVER : " + clientName + " has left the chat");
}
}

public void closeEverything(Socket socket, BufferedReader bufferedReader, BufferedWriter bufferedWriter) {
    removeClientHandler();
    try {
        if (socket != null) {
            socket.close();
        }
        if (bufferedReader != null) {
            bufferedReader.close();
        }
        if (bufferedWriter != null) {
            bufferedWriter.close();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

### 3. Client 클래스

1. #CREATE와 #JOIN을 성공할 때 까지 명령어를 입력하도록 했다.
2. 인자로 두 포트 번호를 받아 저장해놓고, #CREATE나 #JOIN을 할 때 Client 객체를 만들어준다.
3. Client 생성자에서 가장 첫 줄에 `this.socket = new Socket(ip, port);`로 소켓을 만든다. 사용자 입력으로부터 주어진 cmd[]로 cmd이름, 채팅방이름, 클라이언트이름을 지정해준다.

```

public Client(String ip, int port, String clientCommand, String chatRoomName, String clientName) {
    try {
        this.socket = new Socket(ip, port);

        this.bufferedWriter = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        this.bufferedReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        this.clientCommand = clientCommand;
        this.chatRoomName = chatRoomName;
        this.clientName = clientName;

    } catch (IOException e) {
        closeEverything(socket, bufferedReader, bufferedWriter);
    }
}

public static void main(String[] args) throws IOException {
    String serverIPaddress = args[0];
    int portNo1 = Integer.parseInt(args[1]);
    int portNo2 = Integer.parseInt(args[2]);

    Socket socketForFile;

    BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

    Client client = null;
    String[] cmd;
    while (true) {
        // 채팅방 입장
        while (true) {
            // 입력 받기
            String sentence = inFromUser.readLine();
            // 다양한 명령어들
            if (sentence.charAt(0) == '#') {
                cmd = sentence.split("\\s");

                if (cmd[0].equals("#CREATE") || cmd[0].equals("#JOIN")) {
                    // 이름 겹치는 거 있으면 안 됨
                    client = new Client(serverIPaddress, portNo1, cmd[0], cmd[1], cmd[2]);
                    if (client.sendClientInfo()) break;
                    else client.closeEverything(client.socket, client.bufferedReader, client.bufferedWriter);
                }
            }
        }
    }
}

```

4. Client 객체를 만들었다면 밑의 `client.sendClientInfo`를 통해 서버에게 클라이언트 정보를 전해줘서 #CREATE와 #JOIN에 적합한지 검사한다.
  - a. 서버로부터 FAIL 메세지를 받았다면 false를, SUCCESS 메세지를 받았다면 true를 반환하며 서버에게서 온 정보를 바탕으로 위의 while문을 빠져나올 수 있을지 검사한다.

- b. 만약 false가 나오면 client를 없애 다시 while의 `inFromUser.readLine()` 으로 또 입력을 받아 검사한다.

```

public boolean sendClientInfo() {
    if (!socket.isClosed()) {
        try {
            // 서버한테 전송
            bufferedWriter.write(clientCommand + " " + chatRoomName + " " + clientName);
            bufferedWriter.newLine();
            bufferedWriter.flush();

            // 서버한테 받음
            String fromServer = bufferedReader.readLine();
            System.out.println(fromServer);
            String[] fromServers = fromServer.split(":");
            if (fromServers[0].equals("SUCCESS")) {
                return true;
            }
            else return false;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return false;
}

```

## 5. #CREATE와 #JOIN 성공 시에만 메세지를 받는 스레드를 실행시킨다

- EXIT 일 때만 while을 탈출하여 다시 채팅을 시작할 수 있게 해준다. 그래서 가장 바깥 while 안에 두가지 while이 있다. 첫번째는 #CREATE와 #JOIN 성공 시 탈출, 두번째는 #EXIT 시 탈출이다.
- #STATUS를 입력했을 때 이 명령어를 서버에게 전달하여 원하는 정보를 클라이언트가 얻을 수 있도록 한다.

```

// 서버로 부터 내용 받는 스레드 실행
client.listenForMessage();

// 파일 전송 명령어
while (true) {
    String str = inFromUser.readLine();

    if (str.charAt(0) == '#') {
        String[] strs = str.split("\\s");

        if (strs[0].equals("#STATUS")) {
            client.sendCMD(str);
        }
    }
}

```

## 6. #PUT : 클라이언트가 put으로 서버에게 파일은 전송한다.

- 여기서 socketForFile을 만들어줘서 서버 코드의 accept 부분과 연결 될 수 있도록 한다.
- 64000바이트를 기준으로 "#"를 출력하며 스트림들을 닫아주고, 파일을 잘 보냈다는 메세지를 출력해준다.

```

else if (strs[0].equals("#PUT")) {
    client.sendCMD(str);
    // #PUT (FileName) 파일의 송신자는 #PUT 명령어를 이용해서 Server로 파일을 전송
    String fileName = System.getProperty("user.dir") + "/src/tcp/" + strs[1];

    socketForFile = new Socket(serverIPaddress, portNo2);
    FileInputStream fileInputStream = new FileInputStream(fileName);
    OutputStream fileOutputStream = socketForFile.getOutputStream();

    int l;
    byte[] buffer= new byte[64000];

    while ((l = fileInputStream.read(buffer)) != -1) {
        fileOutputStream.write(buffer, 0, l);
        fileOutputStream.flush();
        System.out.printf("#");
    }
    System.out.println("close 하기 전");
    fileInputStream.close();
    socketForFile.close();
    fileOutputStream.close();
    System.out.println("file send success");
}

```

```
}
```

## 7. #GET : 으로 서버로부터 파일을 요청할 것이다

- get을 통해 얻은 파일임을 구분하기 위해 new라는 문자를 붙여 fileName을 만들어준다
- 마찬가지로 스트림들을 닫아주고 잘 받았다는 메세지를 출력한다.

```
else if (strs[0].equals("#GET")) {  
    client.sendCMD(str);  
    // #GET (FileName) 파일의 수신자는 Server로부터 전달 받은 파일 이름을 통해  
    // #GET 명령어를 수행하고 Server로부터 파일을 다운로드  
    String fileName = System.getProperty("user.dir") + "/src/tcp/new" + strs[1];  
  
    socketForFile = new Socket(serverIPaddress, portNo2);  
    FileOutputStream fileOutputStream = new FileOutputStream(fileName);  
    InputStream fileInputStream = socketForFile.getInputStream();  
  
    int l;  
    byte[] buffer = new byte[64000];  
  
    while ((l = fileInputStream.read(buffer)) > 0) {  
        fileOutputStream.write(buffer, 0, l);  
        System.out.printf("#");  
    }  
  
    fileInputStream.close();  
    socketForFile.close();  
    fileOutputStream.close();  
    System.out.println("file receive success");  
  
}
```

## 8. #EXIT 일 때 sendCMD로 서버에게 요청하면 서버는 해쉬맵에서 이 클라이언트를 찾아 remove한다.

- 그리고 close를 해줘서 다시 채팅을 시작할 때 새 클라이언트 객체를 만들어준다.
- #로 시작되는 명령이 아니면 메세지로 간주한다.

```
else if (strs[0].equals("#EXIT")) {  
    client.sendCMD(str);  
    client.closeEverything(client.socket, client.bufferedReader, client.bufferedWriter);  
    break;  
} else {  
    exit(1);  
}  
} else {  
    // 메세지 전송  
    client.sendMessage(str);  
}
```

## 9. 각종 함수들

9-1. sendCMD는 명령어를 보낼 때 쓰인다.

9-2. listenForMessage() 는 서버가 broadcast로 메세지를 보낼 때 읽어들일 스레드이다

9-3. sendMessage()는 클라이언트가 채팅방에서 일반적인 메세지를 보낼 때 쓰이며 FROM 과 함께 클라이언트 이름이 같이 전송된다.

```
public void sendCMD(String str) {  
    if (!socket.isClosed()) {  
        try {  
            bufferedWriter.write(str);  
            bufferedWriter.newLine();  
            bufferedWriter.flush();  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

```

        }
    }

    public void listenForMessage() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                String msgFromGroupChat;

                while (!socket.isClosed()) {
                    try {
                        // 서버로 부터 온 것 읽어들임.
                        msgFromGroupChat = bufferedReader.readLine();
                        System.out.println(msgFromGroupChat);
                    } catch (IOException e) {
                        closeEverything(socket, bufferedReader, bufferedWriter);
                    }
                }
            }
        }).start();
    }

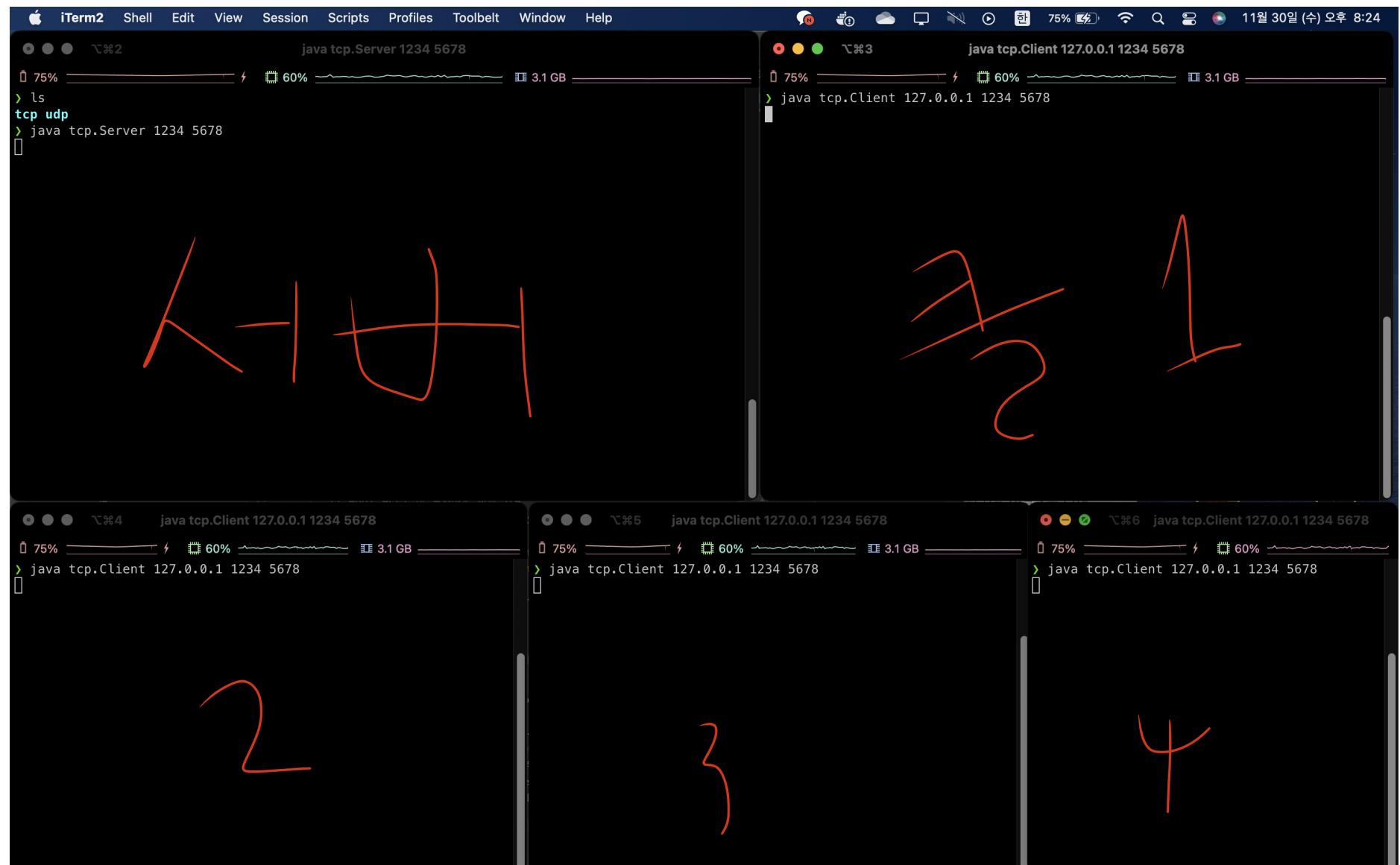
    public void sendMessage(String message) {
        try {
            if (!socket.isClosed()) {
                bufferedWriter.write("FROM " + clientName + " : " + message);
                bufferedWriter.newLine();
                bufferedWriter.flush();
            }
        } catch (IOException e) {
            closeEverything(socket, bufferedReader, bufferedWriter);
        }
    }
}

```

## 5. 실행 예시

### 0. 프로그램 실행 및 채팅방 시작

가장 처음 작성한 방법으로 채팅방 서버와 클라이언트를 준비한다



위의 화면이 나오면 채팅방이 모두 준비되었다.

## 1. JOIN / CREATE 명령 처리 잘 되는지

1-1) 클라이언트1에 JOIN을 먼저하면 실패이다. CREATE로 새 채팅방을 만들어주자.

```
iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
java tcp.Server 1234 5678
ls
tcp udp
java tcp.Server 1234 5678
New client connected!
null
New client connected!
[]

java tcp.Client 127.0.0.1 1234 5678
#JOIN cnet aa
FAIL: chatroom [ cnet ] doesn't exists.
#CREATE cnet aa
SUCCESS: chatroom [ cnet ] is successfully created.

java tcp.Client 127.0.0.1 1234 5678
java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
[]
```

1-2) 클라이언트2에서 #CREATE cnet을 하면 실패이다. JOIN으로 클라이언트1이 만든 채팅방에 들어가자.

```
iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
java tcp.Server 1234 5678
ls
tcp udp
java tcp.Server 1234 5678
New client connected!
null
New client connected!
null
New client connected!
[]

java tcp.Client 127.0.0.1 1234 5678
#JOIN cnet aa
FAIL: chatroom [ cnet ] doesn't exists.
#CREATE cnet aa
SUCCESS: chatroom [ cnet ] is successfully created.

java tcp.Client 127.0.0.1 1234 5678
java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
#CREATE cnet bb
FAIL: chatroom [ cnet ] is already exists.
#JOIN cnet bb
SUCCESS: client [ bb ] successfully joins chat room [ cnet ].
```

### 1-3) #STATUS로 확인하면 잘 보인다.

The screenshot shows four iTerm2 windows running on a Mac OS X desktop. The top-left window (Terminal 2) is the Server, displaying log messages for new client connections. The other three windows (Terminals 3, 4, and 5) are clients. Terminal 3 shows a failed #CREATE command because the chatroom already exists. Terminal 4 shows a successful #CREATE command for chatroom 'cnet'. Terminal 5 shows a successful #JOIN command by client 'bb' into the 'cnet' room. Terminal 6 shows a #STATUS command which returns the room name and client list ('chatRoomName : cnet, clients [ aa bb ]').

```
iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
java tcp.Server 1234 5678
ls
tcp udp
java tcp.Server 1234 5678
New client connected!
null
New client connected!
New client connected!
null
New client connected!
[]

java tcp.Client 127.0.0.1 1234 5678
#JOIN cnet aa
FAIL: chatroom [ cnet ] doesn't exists.
#CREATE cnet aa
SUCCESS: chatroom [ cnet ] is successfully created.
#STATUS
chatRoomName : cnet, clients [ aa bb ]
[]

java tcp.Client 127.0.0.1 1234 5678
#CREATE cnet bb
FAIL: chatroom [ cnet ] is already exists.
#JOIN cnet bb
SUCCESS: client [ bb ] successfully joins chat room [ cnet ].
#STATUS
chatRoomName : cnet, clients [ aa bb ]
[]

java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
[]

11월 30일 (수) 오후 10:06
```

### 1-4) 클라이언트2가 채팅방에서 #EXIT으로 나가면 클라이언트1는 2가 나갔음을 알 수 있고, #STATUS를 확인하면 한 명밖에 없다.

The screenshot shows four iTerm2 windows. The Server (Terminal 2) logs new client connections. The first Client (Terminal 3) creates a room and joins it. The second Client (Terminal 4) also creates a room and joins it. When Client 2 sends a #STATUS command, it shows two clients in the room. Then, Client 2 sends a #EXIT command, leaving the room. When Client 1 sends a #STATUS command, it shows only one client remaining in the room.

```
Notion 파일 편집 보기 창 도움말
java tcp.Server 1234 5678
ls
tcp udp
java tcp.Server 1234 5678
New client connected!
null
New client connected!
New client connected!
null
New client connected!
null
[]

java tcp.Client 127.0.0.1 1234 5678
#JOIN cnet aa
FAIL: chatroom [ cnet ] doesn't exists.
#CREATE cnet aa
SUCCESS: chatroom [ cnet ] is successfully created.
#STATUS
chatRoomName : cnet, clients [ aa bb ]
SERVER : bb has left the chat
[]

java tcp.Client 127.0.0.1 1234 5678
#CREATE cnet bb
FAIL: chatroom [ cnet ] is already exists.
#JOIN cnet bb
SUCCESS: client [ bb ] successfully joins chat room [ cnet ].
#STATUS
chatRoomName : cnet, clients [ aa bb ]
#EXIT
[]

java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
[]

java tcp.Client 127.0.0.1 1234 5678
[]

11월 30일 (수) 오후 10:08
```

1-5) 다시 클라이언트2가 채팅방에 들어왔을 때 클라이언트1에서 #STATUS를 치면 2명이 된다.

The image shows four terminal windows on a Mac OS X desktop. The top-left window is titled 'java tcp.Server 1234 5678'. It logs 'New client connected!' multiple times. The top-right window is titled 'java tcp.Client 127.0.0.1 1234 5678'. It sends '#JOIN cnet aa', receives a failure message, creates the room, and then sends '#STATUS' which returns two clients. The bottom-left window is titled 'java tcp.Client 127.0.0.1 1234 5678'. It creates a room named 'cnet' and joins it. The bottom-right window is titled 'java tcp.Client 127.0.0.1 1234 5678'. It joins the same room and receives the status update from the server.

```
java tcp.Server 1234 5678
> java tcp.Client 127.0.0.1 1234 5678
#CREATE cnet aa
FAIL: chatroom [ cnet ] doesn't exists.
#CREATE cnet aa
SUCCESS: chatroom [ cnet ] is successfully created.
#STATUS
chatRoomName : cnet, clients [ aa bb ]
SERVER : bb has left the chat
#STATUS
chatRoomName : cnet, clients [ aa ]
#STATUS
chatRoomName : cnet, clients [ aa bb ]
```

```
#JOIN cnet aa
#CREATE cnet bb
FAIL: chatroom [ cnet ] is already exists.
#JOIN cnet bb
SUCCESS: client [ bb ] successfully joins chat room [ cnet ].
```

```
#CREATE cnet aa
#JOIN cnet aa
SUCCESS: client [ aa ] successfully joins chat room [ cnet ].
```

```
#CREATE cnet bb
#JOIN cnet bb
SUCCESS: client [ bb ] successfully joins chat room [ cnet ].
```

## 2. 파일 주고 받기가 잘 되는지

2-1) 클라이언트 3, 4가 각각 hyu 채팅방에 입장한다.

The image shows two terminal windows. The left window is titled 'java tcp.Client 127.0.0.1 1234 5678'. It creates a room 'hyu', joins it, and sends a file named 'son.jpeg'. The right window is titled 'java tcp.Client 127.0.0.1 1234 5678'. It joins the 'hyu' room and receives the file from client 3.

```
#CREATE hyu client3
SUCCESS: chatroom [ hyu ] is successfully created.
#PUT son.jpeg
#file send success
```

```
#JOIN hyu client4
SUCCESS: client [ client4 ] successfully joins chat room [ hyu ].
```

2-2) 클라이언트3이 file을 PUT 한다. 그럼 같은 채팅방에 있는 클라이언트4에게 파일이름이 전송된다.

The image shows two terminal windows side-by-side. Both windows have a dark background and a light-colored header bar. The left window is titled 'java tcp.Client 127.0.0.1 1234 5678' and has a status bar at the bottom showing battery level (51%), signal strength, and memory usage (3.1 GB). The right window is also titled 'java tcp.Client 127.0.0.1 1234 5678' and has the same status bar.

**Terminal 5 (Left):**

```
> java tcp.Client 127.0.0.1 1234 5678
#CREATE hyu client3
SUCCESS: chatroom [ hyu ] is successfully created.
#PUT son.jpeg
#####file send success
```

**Terminal 6 (Right):**

```
> java tcp.Client 127.0.0.1 1234 5678
#JOIN hyu client4
SUCCESS: client [ client4 ] successfully joins chat room [ hyu ].
SERVER : [ client3 ] sends file [ son.jpeg ]
```

2-3) 클라이언트4가 그 파일을 GET하면 tcp 폴더에 파일이 new~의 이름으로 잘 들어온다.

The image shows two terminal windows side-by-side, similar to the previous ones. The left window is titled 'java tcp.Client 127.0.0.1 1234 5678' and the right window is titled 'java tcp.Client 127.0.0.1 1234 5678'. Both have a status bar at the bottom.

**Terminal 4 (Left):**

```
> java tcp.Client 127.0.0.1 1234 5678
#CREATE hyu client3
SUCCESS: chatroom [ hyu ] is successfully created.
#PUT son.jpeg
#####file send success
```

**Terminal 3 (Right):**

```
> java tcp.Client 127.0.0.1 1234 5678
#JOIN hyu client4
SUCCESS: client [ client4 ] successfully joins chat room [ hyu ].
SERVER : [ client3 ] sends file [ son.jpeg ]
#GET son.jpeg
#####file receive success
```

